NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN OF A SYSTEM INITIALIZATION MECHANISM

FOR A MULTIPLE MICROCOMPUTER

by

John Lee Ross

June, 1980

Thesis Advisor: R. R. Schell

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Design of a System Initialization Mechanism for a Multiple Microcomputer	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1980
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) John Lee Ross	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June 1980
	13. NUMBER OF PAGES 101
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer System Initialization, Bootstrap Loading, Multiprocessors, Microcomputers, System Generation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents a design for a system initialization mechanism for a multiple processor system. The design is based upon a system of microprocessors (specifically the Intel 8086) being used with a set of application processes, as is common in many real time applications. The design is based upon the concepts ofexplicit communicating processes and explicit memory segmentation- although

DD FORM 1 JAN 73 1473 (Page 1) EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

it does not require full hardware segmentation.

With the goal of simplifying the system initialization function, this thesis segregates the required initialization actions into three distinct phases. The specific phase for each action is determined by which phase provides the most supportive environment for that particular action

While the initialization mechanism described in this thesis was developed for a particular real-time application, the design concepts described are applicable to a variety of hardware and operating system configurations

Approved for public release; distribution unlimited

Design of a System Initialization Mechanism for a Multiple Microcomputer

by

John Lee Ross
Captain, United States Air Force
B.S., University of Missouri, 1973

Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1980

ABSTRACT

This thesis presents a design for a system initialization mechanism for a multiple processor system. The design is based upon a system of microprocessors (specifically the Intel 8086) being used with a set of application processes, as is common in many real-time processing applications. The design is based upon the concepts of explicit communicating processes and explicit memory segmentation- although it does not require full hardware segmentation.

With the goal of simplifying the system initialization function, this thesis segregates the required initialization actions into three distinct phases. The specific phase for each action is determined by which phase provides the most supportive environment for that particular action.

While the initialization mechanism described in this thesis was developed for a particular real-time application, the design concepts described are applicable to a variety of hardware and operating system configurations.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to express my appreciation to my thesis advisor, Lt. Col. Roger R. Schell, for his expert advice and counsel. His comments and suggestions have greatly improved the content and presentation of this thesis.

My thanks also go out to Professor Uno R. Kodres for the time and effort he spent in reviewing my thesis drafts.

A collective "Thank You" to Professor Tien F. Tao, and the staff of the Naval Postgraduate School Solid State Laboratory. Their advice and assistance in hardware-related areas was invaluable during my research effort.

Most of all, I want to express my sincere thanks to my wife, Susan, for her understanding and encouragement, and for the sacrifices she made in support of my graduate studies. Without her, no amount of assistance or technical expertise could have brought this thesis to completion.

I. INTRODUCTION

A. OBJECTIVES

System initialization is the method used to get an operating system loaded and running on a computer system. This is a recurring requirement that must be accomplished each time the computer is powered up and each time the user wishes to change from one operating system to another. This thesis presents a versatile, simple to understand, and widely applicable system initialization mechanism based on a careful sequencing of the initialization activities. These activities will be performed in one of the three system initialization phases addressed in this thesis based upon which phase provides the most supportive environment for each particular activity.

Traditionally, operating system designers have ignored the system initialization problem until the final development stages. As a result, most existing system initialization schemes are rather ad-hoc, using a mass of "special case" activities to accomplish initialization. This thesis addresses these problems by providing a framework for a simple system initialization process that can be used with a variety of hardware and operating system configurations. The approach in this thesis is to make the system initialization mechanism appear as much like a normal

applications program as possible, and thus use the operating system services to the fullest extent. This approach is made possible by two operating system concepts that are being used in many current operating systems on large mainframe and minicomputers, but have only recently been introduced in the microprocessor arena. The first is the concept of segmented memory. The second is the concept of asynchronous processes, including an "idle process" so that the system always "comes to rest" in a state that is easily created and controlled. These two concepts permit the initialization mechanism to avoid the special cases and ad-hoc methods used in so many existing mechanisms.

B. MOTIVATION

For several years, the Solid State Laboratory at the Naval Postgraduate School has been conducting research in the image processing area. A relatively recent area of research has been in the development of "smart sensors" for missile guidance, radar, surveillance, and other image processing applications [1]. Current sensor platforms relay massive amounts of raw data to ground-based processing centers. The smart sensor will provide on-board processing of collected data such that only the initial processed image and periodic updates need be downlinked to the surface. Clearly, a smart sensor will require on-board electronics to do the data processing.

Several Naval Postgraduate School theses, under the supervision of Professor T. F. Tao, have contributed to the development of the smart sensor. In 1977, Yehoshua [2] and Evenor [3] developed filter designs to improve infrared background clutter suppression. In 1978, Hilmers [4] began processing real-world infrared images. All the early computer processing was done on an IBM-360 computer system. In 1979, Celik [5] developed a simulation program on a Digital Equipment Corporation (DEC) LSI-11 microcomputer in an attempt to marry current hardware and software research efforts. Due to its limited primary memory and slow processing speed, however, the LSI-11 proved inadequate for anything but simulation and experimentation. This spawned additional research in the area of microprocessors and microcomputer architecture. In late 1979, Brenner [6] presented a multiple microprocessor system design, using commercially available, off-the-shelf components, that could process the algorithms developed in earlier research and also provide real-time, or near real-time, system response.

Before that goal could be reached, however, an operating system was required to control the operation of the computer system. This operating system would provide an interface between the computer hardware and the user. The operating system concepts used were based on the Multics operating system [13,17]. The basic microcomputer operating system design was developed by O'Connell and Richardson [18]. W. J.

Wasson [7] refined and implemented the basic core, or kernel, of the operating system. The system initialization design presented in this thesis was developed concurrently with the kernel of the operating system.

C. TERMS EXPLAINED

In order to facilitate the discussion of system initializtion, a few terms should be clearly understood.

1. Operating System

The operating system is that set of program modules within a computer system that govern the utilization of computer resources [8]. These resources can be grouped into four major categories: processors, memory, external Input/Output (I/O) devices, and the secondary storage that contains the programs and data.

2. Process

This thesis will refer to the word "process" as the internal representation of a computational task. Each process can be uniquely characterized by its execution point (viz., the state of its processor registers), and its address space (viz., the memory accessible to that process). Since only one process can be running on a physical processor at a time, the operating system will multiplex a number of processes onto each processor. While one process is running, the other processes will be waiting their turns to be scheduled and run. But, when viewed in the long term,

each process can be seen as proceeding through its execution [9]. This is consistent with Saltzer's definition of a process as a program in execution on a pseudo-processor [10].

3. Hardware Configuration

The hardware configuration is defined as that set of hardware components, or modules, present in the system. For example, processors and memory modules are parts of the hardware configuration.

4. Software Configuration

The software configuration is made up of the processes, system tables, and system parameters. For example, the number of processes allowed in the system at a time would be considered a part of the software configuration.

5. System Configuration

The system configuration will be the combination of the hardware configuration and the software configuration.

6. Application

An application is defined as a program that causes the computer system to perform some useful work.

7. Virtual Environment

A key concept in this thesis is that of the virtual machine environment. Briefly, virtualization results in a hierarchy of levels of abstraction, each building upon the facilities provided by the previous level. If the computer

hardware is considered as the lowest level, then the traffic controller, or processor scheduler, could be the next higher level and the applications programs could be the highest level. Thus each level of abstraction runs on the virtual machine provided by the lower levels of abstraction, and each level becomes a part of the virtual machine seen by higher levels.

8. Core Image

A core image will be described as an exact representation of a sequence of instructions and their associated data structures exactly as they would appear in primary memory just prior to execution, but residing on some secondary storage medium. This term is somewhat of an anachronism, since core memory has been replaced by semiconductor memory in most modern computer systems, but it is descriptive of the concept, and will be used extensively throughout this thesis.

9. System Initialization Phases

In one of the few publications dealing with system initialization, Luniewski [11] views the system initialization functions with respect to three phases, or time periods. This thesis follows that same approach.

a. System Generation Time

The bootload medium (viz., a core image of the operating system) is created at system generation time. This normally occurs during a previous session of system

operation, or is done on a seperate development computer system.

b. Bootload Time

Bootload time is when the lowest level of the operating system is actually loaded into the primary memory and its system parameters and tables initialized.

c. Run Time

The period following bootload time, when the operating system programs are running normally, is called run time.

10. Multiprogramming

This term describes a system in which two or more processes can be in one of several "states of execution" at one time. A process is in a state of execution if it has been started but has not yet been completed or terminated by an error condition [8]. In this thesis, a process is said to be "running" if it is assigned a physical processor and its instructions are being executed. A process is "ready" if it could run, but is not currently assigned a physical processor. A process is "blocked" if it is waiting for some event to occur (e.g., an I/O operation to complete or the completion of some action by another process).

11. Multiprocessing

This term implies that more than one processing unit is present in the hardware configuration.

Multiprocessing is used to achieve greater processing power, reliability, and economies of scale.

12. The Bootload Program

A bootload program is a simple program written to run on bare hardware. The bootload program is typically stored in read-only memory (ROM), although it may be extended by a "bootstrap" program read in from a fixed location in secondary storage. It is used to read the core image of the base layer of the operating system from secondary storage, load it into the computer's primary memory, and get the operating system running.

13. The Loader Process

The loader process is one of the modules that are loaded in with the base layer of the operating system. It is similar in function to the bootload program, but it is used to load the higher layers of the operating system and the application programs. The primary difference is that the loader process is used at run time, and makes use of the operating system functions and services provided by the base layer.

D. GENERAL DISCUSSION

In general, the objective of system initialization is to get the operating system loaded into primary memory and running so that it can provide the support facilities necessary to run applications programs. This procedure is

carried out in three basic steps that correspond to the three system initialization phases above. First of all, the bootload program and the core image of the operating sysyem are developed. This phase occurs prior to, and somewhat independent of, the next two steps.

The bootload program is executed in phase two of system initialization. Its purpose is to read the base layer of the operating system from some secondary storage medium (e.g., magnetic tape or disc) and to load the data that it reads into primary memory. The primary memory addresses are either determined by the loader or are encoded in the data. The secondary storage medium will contain the operating system code and data structures. This second phase also involves some preprocessing of the core image data in order that the loader may initialize the processor registers and some operating system data structures in preperation for running the operating system programs. For example, the core image, as it exists on secondary storage, contains load addresses and some key processor register values. The bootload program must strip off this information and use it to initialize the registers and data structures as mentioned above. The details of the bootload program will be discussed further in Chapter III.

The last phase of initialization occurs when the bootload progam passes control to the first executable

statement in the operating system code. At this point, the operating system will begin its normal execution.

It is a basic premise of this thesis that actions performed during system generation time or run time are inherently simpler than the same action performed during the bootload phase. Therefore, this thesis takes the position that the entire system initialization process can be greatly simplified if the core image produced in system generation is as complete as possible, thereby reducing the amount of processing required at bootload time. The justification for this line of reasoning should become clear in the following chapter.

With the layered approach to system generation provided by the virtual environment concept, the most difficult task faced in system initialization is the bootloading of the base level of the operating system. Once this has been accomplished, the initialization process can take advantage of the services provided by this base layer to carry out the remainder of its activities. As subsequent layers are initialized, more and more services become available and the virtual machine seen by the system initialization process becomes increasingly powerful.

E. HIGH LEVEL LANGUAGE PROGRAMMING

Since simplicity and general applicability are two goals of this thesis, the design described herein is oriented

almost totally towards a high level programming language (PL/M). The motivation for this decision came from several sources. Nelson [12] reported a three-to-one increase in productivity when a high level language was used instead of assembly language. While the standard deviations he reported were large, the evidence was overwhelmingly in favor of high level languages. Corbato, Saltzer, and Clingen [13] attribute much of the success of the Multics development to the use of a high level programming language (PL/1) and the interactive debugging that Multics provided. Brooks [14] agrees that the increases in productivity and debugging speed are overwhelming reasons to use a high level language in the design and implementation of systems programs. A high level language will also serve as a communication tool for anyone who reads the program listing. The logical structure of the program can be reflected in the listing, and comments may be inserted at will to clarify potentially confusing portions of the program.

F. STRUCTURE OF THE THESIS

With this chapter as an introduction, Chapter II will present an overview of the environment in which this design was developed and implemnted. This overview will incude the hardware used in the project and a brief look at the philosophy used in the development of the operating system. Chapter III presents the detailed design and proposed

implementation. Chapter IV presents the conclusions reached during the design of this system initialization mechanism, and some recommendations for future research that might use this design as a base.

G. SUMMARY

This chapter has provided the reader with the objectives that this thesis hopes to accomplish, and with the motivation behind the thesis project. It has introduced the reader to system initialization by defining some of the terms used in the thesis, and by presenting a brief general discussion of the initialization function. This chapter has also explained the motivation behind the almost-exclusive use of high-level language programming in the development of the programs for this thesis.

II. THE DEVELOPMENT ENVIRONMENT

A. OBJECTIVE

This chapter will provide a detailed description of the environment in which the system initialization mechanism was developed. It will include an explanation of the hardware used to develop the design for the mechanism, some basic concepts from the operating system it is designed to initialize, and some of the assumptions made about the multiple microcomputer system and the smart sensor algorithms that the system is designed to run.

B. HARDWARE

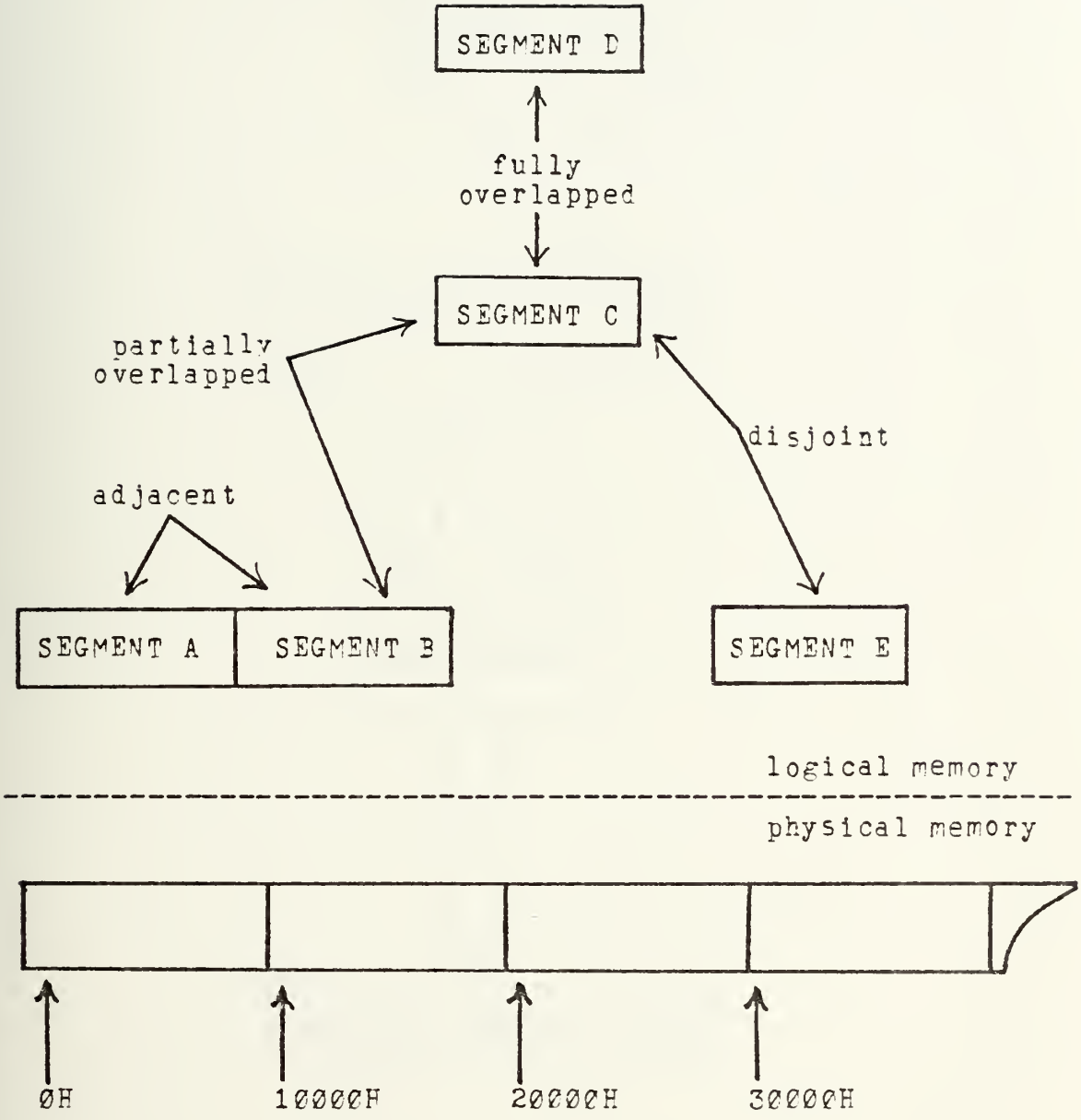
As discussed in the background section of Chapter I, when it was determined that the single LSI-11 microcomputer would handle the processing requirements for a smart sensor system, but would not achieve the desired speeds, the search for a replacement processor suitable for use in a multiple-processor computer system began. The decision was made to focus the search on currently available commercial hardware, since several other reseach activities were exploring the use of specialized hardware for image processing applications. Clock speed, memory size, the number of address and data bits, the bus structure, documentation, and availability were among the primary

selection criteria considered. The search initially identified the DEC LSI-11/23, the Intel 8086, the Motorola 68000, and the Zilog Z8000 as candidates.

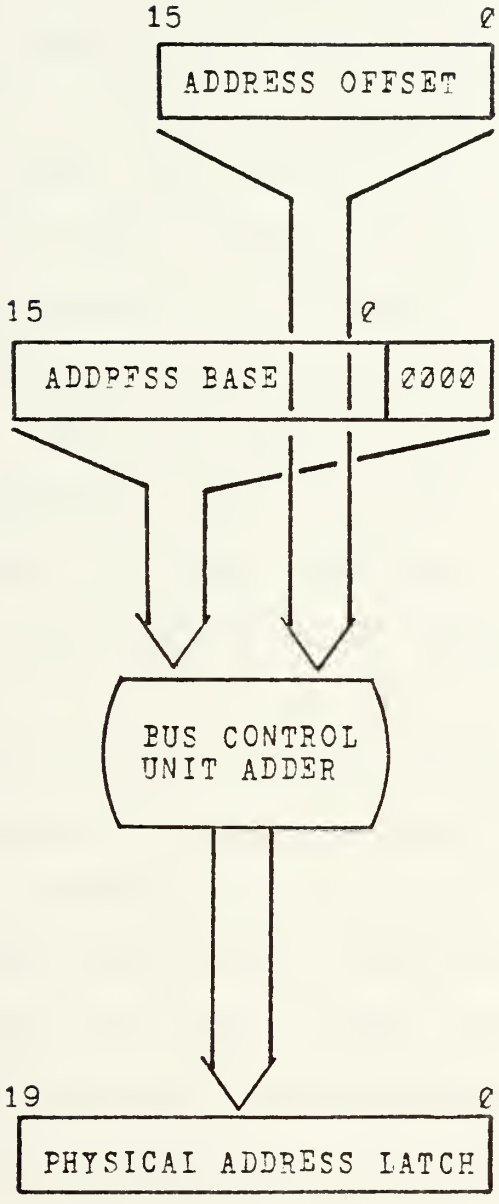
The decision to use the Intel 8086 was finally made, based upon its performance specifications, past experience with other Intel products, and the fact that it was commercially packaged for multiprocessor applications. The fact that it was available off-the-shelf and supported with a full product line of support software and peripheral equipment also had an impact on the selection.

The Intel 8086 is a 16-bit, HMOS technology microprocessor. It has a clock rate of 5 Megahertz (MHz). By combining a base address with an offset, it can directly access a full Megabyte of primary memory. It is capable of both 8-bit and 16-bit signed or unsigned arithmetic in binary or decimal bases, including multiply and divide [15]. It achieves its relatively high speed through a combination of its HMOS technology and some architectural advancements. A major factor in its architecture is the overlapping of instruction fetch and instruction execution. An instruction stream byte queue provides for pre-fetching up to six bytes of instruction during the execution of previously fetched instructions. The exact number of instructions prefetched is a function of the instructions being fetched, since they vary in length.

The one megabyte memory accessible to the 8086 is viewed as a group of segments that are defined by the application. A segment can be described as a logical unit of memory that may be up to 64 kilobytes long [15]. Note that the segment length boundary is not enforced by the hardware. Effective address calculations are done with modulo 64k addition, so attempts to access past this boundary result in "wrap-around" to the beginning of the segment. Each segment is a set of contiguous locations and is an independent, separately addressable unit. As seen in figure II-1, at the hardware level segments may be totally disjoint, adjacent, partially overlapped, or fully overlapped. However, the integrity of this operating system design demands that two segments of a process can never overlap. To access a particular memory location, it is necessary to provide the base address (viz., in a processor base register) of the segment that contains that location, and the offset from the base address to that location. The base address must be an even multiple of 16. To obtain the effective address, given the base and offset, the 8086 performs a left shift of four places on the base address, zero-filling from the low-order end. This shifted base register value is the added to the address offset. This results in a 20-bit effective address, and hence the one megabyte address space. Figure II-2 represents the address-formation process.



Hardware Segmentation in the 8086
Figure II-1



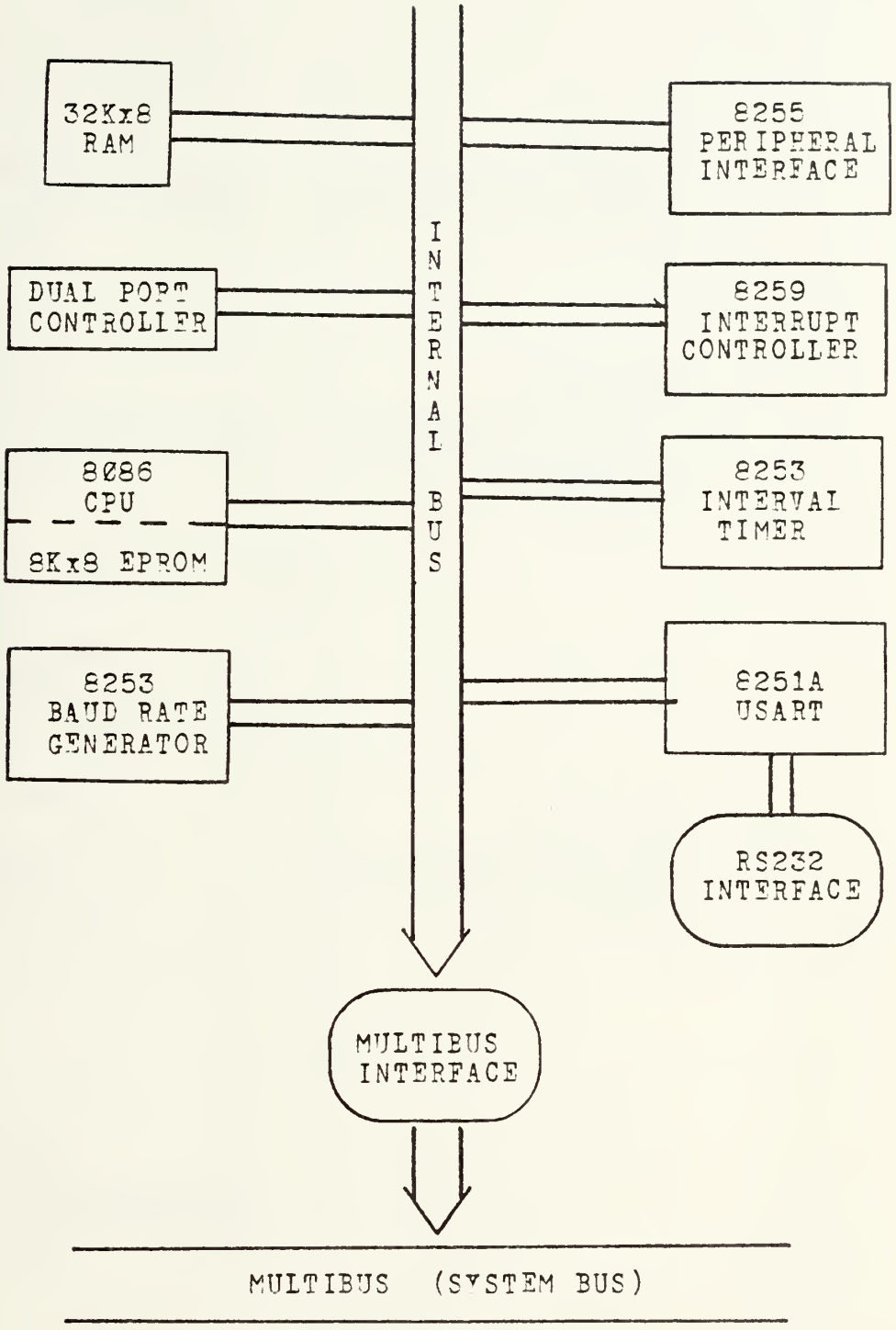
Address Formation in the 8086
Figure II-2

The processor has direct access to four segments at any one time [15]. Their base addresses, or starting locations are contained in four segment registers. The Code Segment (CS) register points to the base of the code segment, from which instuctions are fetched. The value contained in the Instruction Pointer (IP) register gives the offset, from the CS value, to the next instruction to be executed. The Stack Segment (SS) register is a pointer to the base of the stack segment. Stack operations are performed on the locations in this segment. The Data Segment (DS) register points to the current data segment, that is used to maintain program variables. There is also available an Extra Segment (ES) register, that may point to an additional segment used for data storage.

Another major factor in the selection of the Intel 8086 was the availability of the Intel iSBC 86/12A single board computer. The 86/12A is a complete microcomputer system on one 6.75 by 12.0 inch printed circuit board. The version of the 86/12A used in this design contains a 5MHz 8086 processor, 32K bytes of random-access memory (RAM), 8K bytes of electrically progammable read-only memory (EPROM), programmable serial and parallel I/O interfaces, a programmable interrupt controller, a real-time clock, and an interface to the Intel Multibus for interconnection to other devices [15]. At the hardware level, the 32K bytes of RAM is dual-ported. That is, the RAM on one board in a

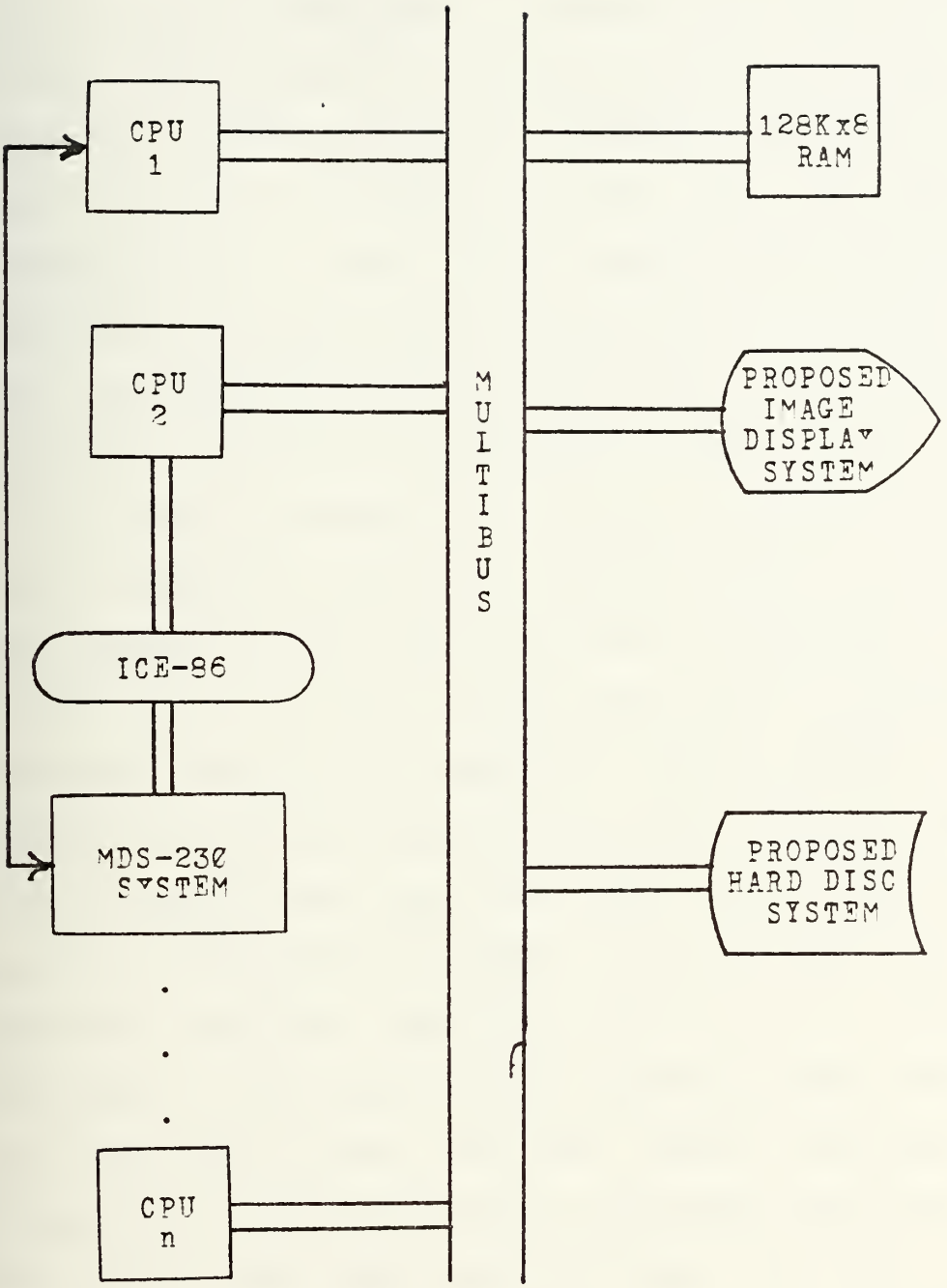
multi-computer system is available to all the other processors in that system. The on-board RAM of each 86/12A is actually seen as two address spaces in a multi-computer configuration. However the operating system design does not support, nor can it tolerate, a segment having two addresses. The dual port feature is used during system initialization, but this is a temporary measure, being used until a suitable bootload program is available in the EPROM. The processor on the same board sees its local memory as the address space between 00000H and 32000H. The other boards in the system see that same RAM as a different address space; the exact address range depends on the board on which it resides and the strapping options employed in the hardware. Figure II-3 shows a system diagram of the iSBC 86/12A single board computer.

The hardware configuration of the multiple microprocessor system used in this thesis project is shown in figure II-4. It is housed in an Intel ICS-80 chassis, which provides the power supplies, cooling fans, and the Multibus connections. System components include a Mu-Pro 128K byte error detecting/error correcting RAM board and up to six iSBC 86/12A's. Near-term hardware enhancements include a Multibus interface to a hard disc system for on-line secondary storage, and an image display device for smart sensor software development.



THE iSBC 86/12A Single Board Computer

Figure II-3



The Hardware Configuration

Figure II-4

Program development was done on an Intel INTELLEC-II microcomputer development system (MDS). Since no secondary storage was available on the multiple microcomputer system, the MDS system was used to simulate secondary storage for the 86/12A's. A program written for the MDS provides communication to one of the multiple microcomputers via a serial-port-to-serial-port connection. The bootload program and the operating system loader view the port just as if it were the interface to a secondary storage device.

As shown in figure II-4, the two computer systems are also connected by an Intel ICE-86 in-circuit emulator [16]. The ICE-86 is used to aid in program development. In this application, it is also used to load into the 86/12A's those programs that will eventually reside in EPROM. Since the 86/12A's do not have direct access to secondary storage via the system bus, the run-time loader process that runs on the processor connected to the MDS via the serial port link must perform the disc I/O function and make the disc data available to the other loader processors. When the hard disc is installed, all the run-time loader processes will be identical. Until that time, the method described above and detailed in the next chapter will be used for system initialization.

C. OPERATING SYSTEM BASICS

The operating system developed for the microcomputer system described above was written by W. J. Wasson [7] in a thesis project that was done concurrently with this thesis. It uses many of the concepts developed for the Multics system [17], and is an extension, with a few changes, of the distributed operating system concepts presented by O'Connell and Richardson [18]. The operating system is intended to provide an interface between the user and the hardware such that the underlying hardware configuration is made invisible, or at least of no direct concern, to the user. This section of the thesis is intended as a basic intoduction to those operating system concepts and mechanisms that directly affect system initialization. The reader is referred to the thesis by Wasson [7] for additional details.

1. Processor Multiplexing

This operating system makes use of the virtual environment concept introduced in chapter one. This concept provides a layered operating system consisting of several levels. At the lowest level is the Inner Traffic Controller, whose function is to multiplex Saltzer's "pseudo-processors" [10] onto the physical processors present in the system. The primary data base used by the Inner Traffic Controller is the Virtual Processor Map. A virtual processor is defined as a "simulation" of a processor using a physical

processor to interpret the instructions "executed" by the simulated processor. This data structure contains the virtual processor execution state, its scheduling priority, interprocess communication information, a descriptor for its address space (represented by the location of its stack segment), and a scheduling flag that signifies that the processor has been sent a virtual preempt interrupt by some other virtual processor.

At the next level is the Traffic Controller. The Traffic Controller serves to multiplex processes onto these pseudo-processors. The data structure used by the Traffic Controller is called the Active Process Table. This table contains the information needed to get a process loaded onto a virtual processor and running.

Wasson also provides a "Gate" module at the next level to simplify the user's interface to the operating system functions by providing a single entry point to the lower levels of the operating system. The programmer interfaces with all operating system functions by making a "call" to the gate module using the parameters for the requested function as arguments in the call.

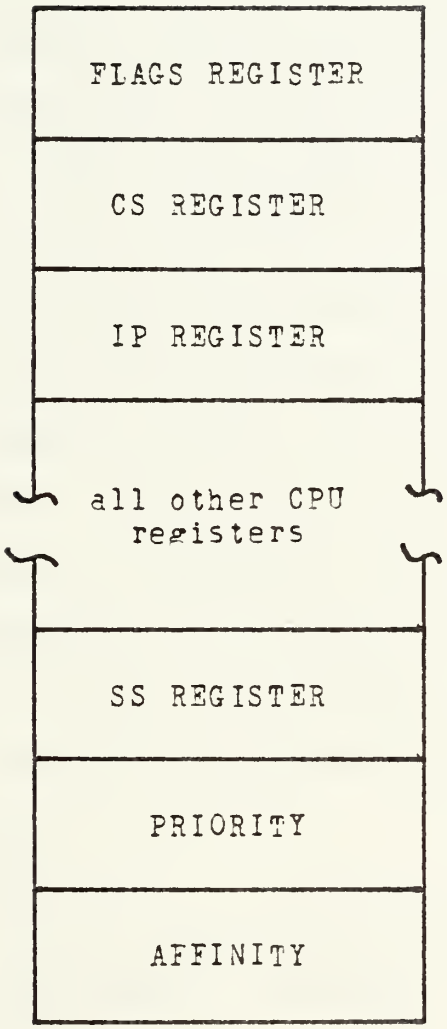
2. The Process Parameter Block

In addition to loading the processes into memory, system initialization must also identify these processes to the operating system so that they can be scheduled and run. The initialization mechanism described in this thesis uses a

Process Parameter Block to pass process definition parameters to the process creation function of the operating system. The Process Parameter Block is a per-processor artifice into which each run-time loader process stores definition parameters for the process being loaded. When the operating system is ready to create [7] the process, it extracts the parameters fron the Process Parameter Block. Since processes are loaded and created one at a time, the memory locations in the parameter block can be reused for each process. As seen in figure II-5, the Process Parameter Block contains values for all the processor registers associated with a process. Only the CS, IP, and SS register values are of concern in this thesis, but the structure was designed to provide easy expansion during later research. The Priority is used by the scheduling algorithm. The Affinity is used to bind a process to a particular processor.

3. Interprocess Communication

Of primary importance to any multiprogramming or multiprocessing system is inter-process communication to synchronize cooperating processes and control access to shared resources. This operating system uses the "Eventcounts and Sequencers" mechanism proposed by Kanodia and Reed [19]. A summary of this mechanism is provided here, since interprocess communication is vital to the run-time loader processes.



Process Parameter Block

Figure II-5

An eventcount is a system variable that represents a class of events that will occur in the system. A virtual processor can perform three primitive operations on eventcounts. It may obtain the current value of an eventcount by performing a READ of that eventcount. It can increment by one the current value of an eventcount by doing an ITC_ADVANCE on that eventcount. Finally, a virtual processor may await the occurrence of a particular event within the class of events associated with an eventcount by doing an ITC_AWAIT on that eventcount. This mechanism can be simply viewed as using a counter to control the virtual processors. However it offers an advantage over the traditional semaphore or mechanism. The occurence of an event can be broadcast to several virtual processors who might be awaiting it. This is more difficult to achieve with more traditional interprocess communication schemes.

D. DEVELOPMENT TOOLS

As mentioned earlier, all program development was done on a seperate development computer system. One major advantage of using such a system is the supportive environment it provides the programmer. This support is in the form of the software development utilities available from the manufacturer of the development system. In the development of the system initialization programs for this thesis, the decision was made to take full advantage of

these utility programs. In addition to the PL/M-86 compiler, three other utility programs, provided by Intel, are used extensively during the system generation phase to create the core image of the operating system to be loaded during the bootload phase. These three Intel programs are called LINK86, LOC86, and OH86 [20]. They are used to perform the functions of linking, locating, and object file transformation. Each of these functions is discussed below. Appendix A contains annotated sample outputs from the development utility programs described in this section.

1. Compiling Program Modules

The PL/M-86 compiler [21], in addition to translating the high-level language statements into 8086 machine instructions, offers four mode options. These options let the programmer determine the degree of segmentation to be used. The SMALL option tells the compiler to produce only two segments. One segment combines the code sections of all the modules in the program (or program section). The other segment contains all the constant and variable data and the stack. This mode provides the greatest run-time efficiency, since the Code Segment register and the Data Segment register (which in this mode is identical to the Stack Segment register) do not change during run-time. The trade-off is that the total size of each of these segments may not exceed 64k bytes, and that there is very little memory allocation flexibility.

At the other extreme is the LARGE compile mode. In this mode, the code section of each module is allocated a separate segment. The same is true for the data section of each module. The stack sections of all modules are combined to form a single stack segment. This mode pairs up the code and data segments of each module and insures that the CS and DS registers always contain the values from the same module. In this mode, the total amount of code and data may exceed 64k bytes, but any one segment is constrained to 64k.

The COMPACT and MEDIUM modes fall in between the two modes discussed, and offer differing degrees of segment seperation. The PL/M-86 Compiler Operator's Manual [21] states that all modules in a program must be compiled in the same mode. To maintain flexibility and to achieve the finest granularity of segment control, the LARGE mode is used on all operating system and application program modules run on the computer system used for this thesis project.

2. Combining Program Modules

LINK86 is a program used to combine the separately developed and compiled program modules into a relocatable object module. When these separate modules were compiled, all addresses were relative to the beginning of each module. LINK86 accepts these separate modules as input, and produces as output a single combined module whose addresses are relative to the beginning of the linked output module. In so doing, it resolves all intermodule references to variables

and procedures. The availability of the linker permits the programmer to develop small, managable program modules that can be debugged and maintained separately, and then bound into a single module prior to loading.

3. Assigning Memory Locations

The LOC86 program takes as input the relocatable object module from the linker and produces as output an absolute object module in which all addresses have been converted to physical memory locations. It also produces a memory map which reflects the binding performed and a symbol table that shows the memory location assigned to each variable, label, and procedure. LOC86 also allows the user to specify exactly where in memory he wants the various modules of his program to be located.

4. Object to Hexadecimal File Conversion

The output of the locator is an absolute object file of the input. This object file, as it exists on secondary storage, is a sequence of binary digits. Encoded in this sequence of binary digits are all the machine instructions and data necessary to run the process. Before execution can actually take place, however, certain key processor registers (viz., the code segment, instruction pointer, and stack segment registers) must be initialized to their proper values. This is one of the responsibilities of the initialization mechanism. These values are contained in the binary object file. For the equipment used in this thesis,

the exact format of the data in these object files was not presented in any documentation available from the manufacturer. Before the initialization mechanism can perform any programmed action on the object files, it must have, or be able to ascertain, the file format. Fortunately, there is a file conversion program, called OH86, which converts this binary object file to the hexadecimal ASCII format. This program, and the output file it produces, is well documented. In an effort to expedite development of the initialization mechanism, it was decided to use the OH86 program and convert the object files to ASCII, so that they could more easily manipulated.

There is, however, a storage space trade-off to consider. For example, the eight-bit binary value, 0100 1111, is read as 4F in hexadecimal. To encode this in ASCII, one byte is required for the ASCII representation of the 4(0011 0100), and one byte is required for the F(0100 0110). This representation scheme requires twice as much storage in the MDS as the binary form, but because of limited documentation it makes the development of the initialization mechanism much simpler. The bootstrap program and the loader process in this thesis contain a simple procedure which converts this ASCII representation back to binary before storing the data, so there is no waste of memory in the multiple microcomputer system.

E. ASSUMPTIONS

In an effort to expedite work on the algorithms for the smart sensor, several assumptions were made which would simplify the design of the initialization mechanism and the operating system. This simplification primary involves the allocation and partial completion of some operating system tables used at run time. These tables are used to describe to the operating system the set of processes that will be running, and the hardware configuration that it will be running on. In a general-user computer system, some of these assumptions might not be valid. Future systems programs developed for the multiple microcomputer system may wish to generalize the system initialization mechanism to eliminate some of these assumptions.

The key assumption made is that the run-time environment is very static. That is, the set of processes to be run and the hardware configuration is known at system generation time, and remains constant during run time. This assumption is justified by the fact that the algorithms to do the processing experiments for the smart sensor system can be partitioned into processes before the actual processing is done. Therefore, a lot of information about these processes can be determined during system generation and passed to the bootload and execution phases. For example, all the processes that will be executed at run time can be identified at system generation time.

Luniewski[11] also states that in order to simplify initialization and still permit dynamic reconfiguration [9], some minimal hardware configuration should be assumed by the initialization mechanism. This is intuitive, since without at least one processor and some amount of primary memory, a computer can do no useful work. Given this minimal hardware configuration, that is a subset of the largest potential hardware configuration, the initialization mechanism could employ dynamic reconfiguration to establish the actual hardware configuration. In an effort to maintain simplicity, this thesis does not attempt to implement dynamic reconfiguration. Instead, the hardware configuration assumed by the initialization mechanism is the full set of hardware present in the system. Since fault-tolerance, which requires the capability to dynamically reconfigure the system, is a long-term goal of the smart sensor program, continuing research is being carried out to give this initialization mechanism that capability.

These assumpmtions permit linking and locating of the user's modules with the same justification as is used for the operating system modules - they do not change during the lifetime of one initialization. Thus they can be treated the same as the system processes, and their linking and locating can be performed during system generation. While this approach is contrary to the accepted practice of delaying the binding of logical resources (viz., memory segments) to

physical resources (viz., memory locations), to enhance system flexibility, it is fully justified in this application by the fact that the environment is stable.

The most important item of information that this assumption provides is a partial definition (viz., the address space) of each process that will be run. This allows the Process Definition Table, shown in figure II-6, to be created during the systen generation phase. The information in this table includes the process name (used to address its MDS file), its initial CPU registers, its stack base (used for process creation), its scheduling priority, and its processor affinity. Processor affinity implies that the programmer can state which physical processor his process will be run on. This is important in the case of a system with dissimilar processors. For example, one single board computer might be enhanced with a hardware multiplier circuit, or a special-purpose I/O processor. Also included are the initial CS and SS register values. This structure is created from information provided by the programmer who developed each process.

Another important function that can be done at system generation time is the allocation of specific segments to the local on-board memory or to the global shared RAM. O'Connell and Richardson [18] present the design of an automated decision technique for memory allocation. Their design calls for a dynamic memory management scheme. That

Filename	CPU Regs	Stack Base	Priority	Affinity

Process Definition Table

Figure II-6

is, memory allocation and deallocation is a run time function. The mechanism proposed in this thesis performs the same memory allocation tasks, but they are performed during system generation. The global-vs-local decision is based on the two-by-two decision matrix shown in figure II-7, and on a manually- maintained memory map that keeps track of the free and allocated portions of memory. Note that the upper lefthand quadrant of the decision matrix in figure II-7 shows two possible choices for locating shared, non-writeable segments.

While memory can be conserved by locating shared data in global memory to avoid duplication, the choice in this design is based upon the desire to keep as many segments as possible in the local, on-board memory of the using processor. Since each access to global memory requires exclusive use of the system bus for the duration of that access, all other processors who might want to access global memory during this period are forced to wait until the bus is free. For this reason, accesses to global memory should be held to a minimum. This can be accomplished by locating all executable (viz., pure) code and as much data as possible in the local RAM, and using global storage for only those variables and data that are shared and writeable.

	WRITEABLE	NON-WRITEABLE
SHARED	GLOBAL	LOCAL/GLOBAL
UNSHARED	LOCAL	LOCAL

Memory Allocation Decision Matrix

Figure II-7

F. SUMMARY

This chapter has presented the environment in which the design described in this thesis was developed. It has shown the hardware involved, an overview of some important operating system principles, a look at the software development utilities used in system generation, and the assumptions made in the thesis and their implications. With this information as background, the thesis will present, in the following chapter, the design of the initialization mechanism developed for this thesis.

III. THE DESIGN

A. OBJECTIVES

This chapter will examine the different environments in which the three phases of system initialization - system generation, bootloading, and run time - take place. This discussion will unfold the design of the initialization mechanism developed for this thesis. It will also provide the reader some insight into the sequencing of the initialization activities and how the timing of these activities effect the complexity of the initialization process. As this discussion progresses, more and more references will be made to operating system functions and services. The reader desiring more details on the operating system, per se, should refer to the thesis by Wasson[7] for a more complete explanation.

B. OVERVIEW

Chapter I discussed the purpose of system initialization and the three phases of initialization used in this thesis. Recall that during the system generation phase, the bootload medium, a core image of the base layer of the operating system, was created. The other two phases- bootload and run time- perform the loading of this core image as well as the remainder of the operating system and the application

programs from secondary storage into the computer system's primary memory. The initialization mechanism proposed in this thesis involves two seperate loading functions. Recall that the bootload program, which runs on the bare hardware, is used to load the base layer of the operating system into primary memory and start it running. This program is normally ROM resident so that it may be started by activating some hardware "Reset" or "Bootload" switch.

The second loading function is part of the distributed operating system, and is loaded into each processor during the bootload phase along with the base layer of the operating system. This loader is used during run time to load the remainder of the operating system and the application programs and to prepare them to be scheduled and run. This dual-loader approach is common in most existing initialization schemes, and will be discussed in detail later in this chapter.

In this application, since only one processor has access to secondary storage system on the MDS, the run-time loader on this processor is a slightly enhanced version of the loader process that runs on the other processors. These enhancements include a "disc I/O" routine, to allow that loader to access the MDS disc information sent to the 86/12A serial port, and a procedure to check the Process Definition Table to determine when the loading function for this

process is complete. For ease of discussion, this enhanced loader will be referred to as the controlling loader.

C. THE SYSTEM GENERATION SEQUENCE

Before the loading begins, however, there is some preliminary work to be done that will simplify the remainder of the initialization. This work is done during system generation. As discussed in Chapter I, this thesis proposes that actions performed at system generation time or subsequently at run time are inherently simpler than that same action performed at bootload time. This is due to the more supportive environment available at system generation time, and the operating system services available at run time. Compare these to the bare-hardware environment at bootload time, and the reasoning behind this premise becomes clearer. A look at the environment in which system generation takes place will provide additional justification for the proposal.

Since system generation takes place prior to the bootload and execution phases, it enjoys the supportive environment provided by an existing operating system and any available utility and library routines. As mentioned in Chapter II, the program development for this thesis was accomplished an Intel Intellec Microcomputer Development System (MDS). The design proposed in this thesis makes extensive use of the utility programs available in that

environment to accomplish the system generation tasks. System generation also enjoys the luxury of time. The use of the ISIS-II operating system in the MDS serves to reduce the complexity of the bootload and run time phases.

Because of the static nature of the image processing application for which this initialization scheme was designed, the system generation phase can make the assumptions regarding the hardware configuration and the nature of the application programs discussed in Chapter II. These assumptions permit extensive preliminary processing to be done in the more comfortable environment of system generation. This relieves the later phases, which occur in much less supportive environments, of the preparatory processing that they would otherwise be required to perform.

By assuming that the hardware and software configurations are known at at system generation time, that they will remain constant from one initialization to the next, and that dynamic reconfiguration is not an issue, all memory allocation decisions can be made during system generation. As discussed in Chapter II, the decision as to whether a segment should be placed in local or global memory is based on a two-by-two decision matrix. The main difference between the Richardson and O'Connell[18] allocation scheme and the scheme employed in this thesis is that the scheme used here is manual, rather than automated. This means that that memory allocation is a one-time system

generation requirement rather than on on-going run-time function. The O'Connell and Richardson [18] decision matrix and memory map are maintained on paper, by the person generating the system, rather than as data structures maintained by the the system initialization mechanism.

The simplest way to view system generation is as a time-sequence of events, beginning with program design and ending with the creation of the load module, or core image to be loaded. A detailed examination of this sequence of events will provide a foundation for the design choices made throughout the development of the initialization mechanism described in this thesis.

1. Program Design

The operating system and initialization scheme developed for this project rely on the programmer to design his programs to take full advantage of the multiprogramming and multiprocessing capabilities provided by the hardware and the operating system. This requires that the programmer be somewhat, though not intimately, familiar with the operating system philosophy and the hardware configuration. Given this basic knowledge, and the widely-accepted technique of structured programming, it is relatively easy for the programmer to design the required process structure into his programs. This involves partitioning each application into a group of cooperating processes, and including in each process the necessary operating system

calls to provide inter-process synchronization, and explicitly declaring shared memory segments for communication between processes.

In the development of each process, there are some simple "ground rules" the programmer should follow to simplify memory allocation and enhance the performance of the system. First, all data shared by processes should be declared to be in segments which are "external" to the application procedure [22]. This implies that the variable is declared and defined elsewhere. Furthermore, an absolute memory address must NEVER be coded into any application. Second, all program code should be reentrant [22]. This allows each invocation of a procedure to store its variables on the process stack. Thus one invocation will not overwrite the variables used by the previous invocation, as would be the case if the variables were stored as part of the procedure itself. The third ground-rule is imposed to reduce the system bus contention problem discussed in Chapter II, and merely requires that references to shared, writeable variables and structures be held to a minimum. This typically involves a single read reference to "input" data to the process and a single write reference to "output" the data (results). In particular, shared segments should never be used for temporary or intermediate results. The fourth rule requires that the programmer segregate writeable and readable segments whenever possible. This will allow finer

granularity in the memory allocation process. Finally, the programmer must declare the Gate module as an external procedure in every process to be run. This will resolve all the external references to the operating system interface.

The programmer is also given the responsibility of initially identifying his process to the operating system. Recall that a process can be identified by its address space and its execution point. Therefore, the programmer must identify all the segments in the process address space and must identify which of these segments will be modified (written into) by this process. Furthermore, the programmer must identify the initial entry point, and any parameters passed to this entry point. This information is actually provided to the system operator, who prepares the Process Definition Table and makes the memory allocation decisions based on the full set of initial process identification information, as discussed below in the section on memory allocation.

2. Compilation

After the program has been developed and written, it must be compiled. The compiler translates the high-level language code into machine language instructions. For this application, an additional check is made at system generation time to insure that all program modules have been compiled with the same mode option. Recall from Chapter II that the compiler mode option determines the degree, or

granularity, of the segmentation. This information must be supplied by the programmer, since he is the one who performs the compilation.

3. Linking

The third step in the system generation sequence is the linking together of the various modules that make up a process. Since the programmer knows exactly which modules comprise his process, he is in a position to pre-link these modules. Since each process needs an interface to the operating system, each process is also linked to the Gate module previously described. This implies that each process has declared the Gate module as an external procedure.

4. Memory Allocation

While the programmer is in the best position to compile his modules and link them into individual processes, he is not in a position to know the degree of segment sharing that will take place. Neither is he in a position to know where, in the system memory, other programmers might elect to load their processes. Clearly the memory allocation decisions must be centralized to avoid chaos. The computer system operator, or perhaps a "chief programmer", is in the best position to make these decisions. This thesis will assume that these decisions are made by the operator as part of the system generation process. As mentioned in Chapter II, the global-vs-local decisions are made using a decision matrix.

But the decisions as to the specific memory locations to allocate for each segment require some information from the programmer. Specifically, the programmer must provide a list of the segments in the address space of his process, the length of each segment (which is available from the linker output), and whether each segment is writeable or non-writeable. The identification of segments must be unique across all processes in the system to insure that shared segments can be unambiguously distinguished. Figure III-1 shows a suggested Process Information Form which might be used to standardize the content and format of this information. The form contains one entry for each segment in the address space, and indicates which of the above attributes apply. The programmer is also asked to identify which other processes will share each segment. This is used only to cross check for possible design errors in interprocess communication. The per-process list also includes the initial parameters, the process priority, and processor affinity information that the operator needs to build the Process Definition Table used by the bootload program and the run-time loader processes. This information form is provided for each application process and (separately) for the operating system kernel for each physical processor. The kernel includes only one per-process data segment: the kernel stack. Since the kernel is linked only once for each processor, the operator must "create" the

PROCESS INFORMATION LIST

PROCESS NAME: _____ PRIORITY:____ AFFINITY:___

Initial Parameters: SS:_____ AX:_____ BX:_____

CX:_____ DX:_____ ES:_____

INDEX	SEGMENT NAME	LENGTH	READ	RD/WT	SHARING PROCESSES
1					
2					
3					
4					
5					
6					

Process Information Form

Figure III-1

corresponding stack for each process. As discussed by Wasson [7], the kernel stack must be allocated as a logical extension of, and at a lower address than, the stack segment for each process.

Armed with this process information list and the allocation decision matrix, the operator is now prepared to make the actual allocations of specific memory locations to segments. Since he is, in effect, the Memory Manager process described by O'Connell and Richardson [18], he will maintain the System Memory Maps, for both local and global RAM, which reflect the status of the system memory. As shown in figure III-2, the memory map contains the base address and length of each named segment and the base address of the free or unallocated areas of memory. The memory map is completed as a sorted list to aid in detecting allocation errors made by the operator. The local and global memory in the system is allocated separately; only shared, writeable segments are allocated to global memory. A useful guideline is to allocate all local kernel segments at addresses below the applications so that applications stacks can never "overflow" into the kernel. Recall from above that the operator must "add" a kernel stack segment for each process. It is also up to the operator to avoid "checkerboarding", or fragmentation, the condition in which many small free areas exist whose combined sizes are large enough to contain a segment, but none are large enough alone. This condition can

MEMORY MAP

SEGMENT NAME (OR "FREE)	BASE	LENGTH

Memory Map

Figure III-2

usually be avoided by careful allocation, but it may also involve some trial-and-error to obtain a proper fit.

5. Locating

Once all the allocations decisions have been made, the actual assignment of physical memory locations is made using the locator utility program, LOC86. The system operator passes the allocation decisions made for each process to LOC86 as parameters. These parameters indicate to the locator the base address of each segment, including the kernel stack, in the process address space.

The operating instructions for LOC86 contain the options and parameters required to control memory allocation [20]. The output from the locator is the binary core image of the process that was input to it. This image is complete with load addresses for the code and data in the process, as well as the CS and SS register values necessary to start the process running. The locator is run once for each application process, and once per CPU to locate the distributed operating system kernel that is available through the Gate to all processes.

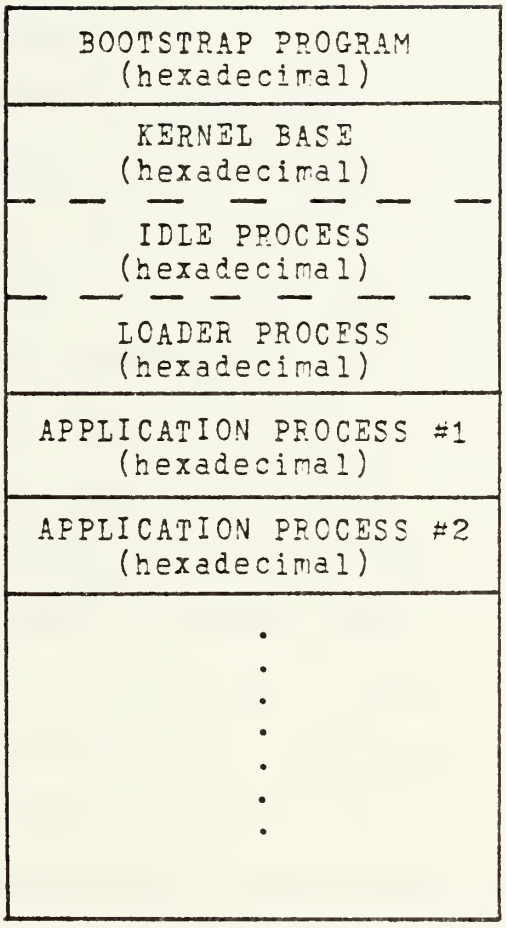
6. File Conversion

As discussed above, the memory management function was not automated due to the lack of documentation concerning binary object files. For the same reason, the bootload program and the run-time loader processes were designed to read the ASCII files output by the file

conversion program, OH86. The OH86 output format is well documented [20]. So the last step in the system generation process is to run OH86, once per located process and CPU kernel, to transform the binary object file into the ASCII format expected by the loading processes. A skeletal example of the output produced by OH86 is contained in Appendix A.

7. System Generation Summary

Before proceeding into a discussion of the bootload phase and the environment in which the bootload program runs, it will be beneficial to pause and examine exactly what was accomplished during system generation, and exactly where the initialization process stands when system generation has been completed. This thesis views system generation as a time-sequence of events that begins during program design, and proceeds through compilation, linking, memory allocation, locating, and file conversion. At this point, the ASCII representation of the core image of each process to be loaded has been created and stored as a file on the secondary storage (viz., floppy disc) in the MDS. The disc also contains two other files: the bootstrap program and the kernel base with the run-time loader process. A graphic representation of the disc, as it appears at the end of system generation time, is shown in figure III-3. Note that for each process the loader needs the disc address (i.e. track number and sector number) of the target file. In the MDS-based loader, this address is the actual



Disc Contents at end of System Generation

Figure III-3

filename, since the filename is used by the ISIS-II operating system disc routines on the MDS. The filename is one of the items of information available to the loader process in the Process Definition Table.

D. THE BOOTLOAD PHASE

When it is desired to initialize the system and run the application programs, the bootload phase begins. In most computer systems, the bootload program is invoked by activating a "reset" or "bootload" switch. This causes a jump to the first instruction of the bootload program, which is contained in ROM. After the proposed hardware enhancements have been made, and the complete operating system has been developed, the bootload program for this system will be placed on EPROM, and will be invoked in this same manner. This section will discuss the sequence of initialization actions that take place upon invoking this ROM-resident bootload program.

Like system generation, the bootload phase can be viewed as a time-sequence of activities, beginning when the bootload switch is pressed, and ending when the operating system kernel is running. When the bootload switch in the multiple microcomputer system is depressed, it causes a hardware interrupt to occur in all the processors in the system. The interrupt handler for the bootload interrupt is the ROM-resident bootload program in each processor.

1. Invoking the ROM-resident Bootloader

The bootload routine is a small, very simple program that serves three basic functions. First of all, it must determine which CPU in the system will be the "Bootload CPU". The Bootload CPU will serve as the master or controlling CPU throughout the bootload and run-time loading phases. While the bootload programs in all CPU's are identical, the Bootload CPU will execute some sequences of instructions that the other processors will not. When the bootload programs begin execution, each one will attempt to read the same variable in global memory. This variable will be initialized by the EPROM programs to a predetermined value. As mentioned in the section on memory allocation, access to global memory requires that a processor have exclusive use of the system bus. There is a built-in system bus "lock" that can be set as soon as a processor gets control of the bus. This lock will be used to resolve the conflict of multiple simultaneous access attempts. The processor that first gets control of the bus will become the Bootload CPU. This processor will then alter the value of the global variable. When the bus lock is turned off, and other processors are able, in turn, to access the variable, they will see that the variable has been altered, and enter a wait loop, awaiting further instructions from the Bootload CPU.

To permit the programmer to specify which physical processor he wants his processes to run on (i.e., the affinity of the process), there must be some way to identify these processors. Physically, the processors can be identified by some unique serial number or identification number. This type of identification is inconvenient for the operating system because the physical processors can be removed and replaced for maintenance, testing, and for various other reasons. Therefore, the initialization scheme needs a method of assigning logical CPU numbers to the physical processors currently in the system. This can be done in a manner similar to determining the Bootload CPU. By convention, this scheme assigns logical CPU number Ø to the Bootload CPU. The Bootload CPU enters its serial number, which is contained in its EPROM, into the first entry of a global structure called the CPU$TABLE. The Bootload CPU then sets a global variable called LOGICALCPUNUM equal to 1, and unlocks the lock which has been associated with that variable. The other processors will now "race" to access LOGICALCPUNUM. The winner of the race will set the lock, enter its serial number into the second entry in the CPU$TABLE, increment LOGICAL$CPU$NUMBER, and then unlock the lock. This process will continue until all the physical processors have been assigned a logical CPU number. The Bootload CPU will know how many CPU's there are in the configuration and that all processors in the system have

been assigned a logical number after some fixed time period (a few milliseconds) has elapsed. In addition to the two CPU numbers in the CPU$TABLE, each processor also has a "mailbox": a location used for a primitive method of interprocessor communication with the Bootload CPU.

2. Accomodating the Initial Hardware

As previously discussed, the hardware configuration does not presently include online secondary storage, and the decision was made not to write the bootload into EPROM until the development was complete. Some temporary alterations were made in the initialization mechanism to permit the devlopment to proceed with this initial hardware configuration. The use of the MDS to simulate secondary storage was mentioned previously. The bootstrap program reads data from the serial port of one of the 86/12A single-board computers. A program was written for the MDS that reads the hexadecimal object files from floppy disc and outputs the hexadecimal data to the MDS serial port. There is a cable connecting the two serial ports. The cable is made to allow a primitive sort of protocol between the two systems via the "clear to send" and "request to send" status lines [23]. This constrains the loading funtion to having access to secondary storage from only one processor, rather than from any processor on the statem bus. To simulate the presence of an EPROM bootload program, the ICE-86 in-circuit emulator was used to load the bootload program into RAM.

Using the dual-port memory capability, the ICE-86 can load the bootload into each processor's local memory. The ICE-86 was also used to alter the interrupt vector in each CPU so that the preempt interrupt would transfer to the bootload program. Finally, the processor connected to the MDS was given a slightly different version of the bootload program that starts its execution by sending a preempt interrupt to all other processors, simulating the bootload switch.

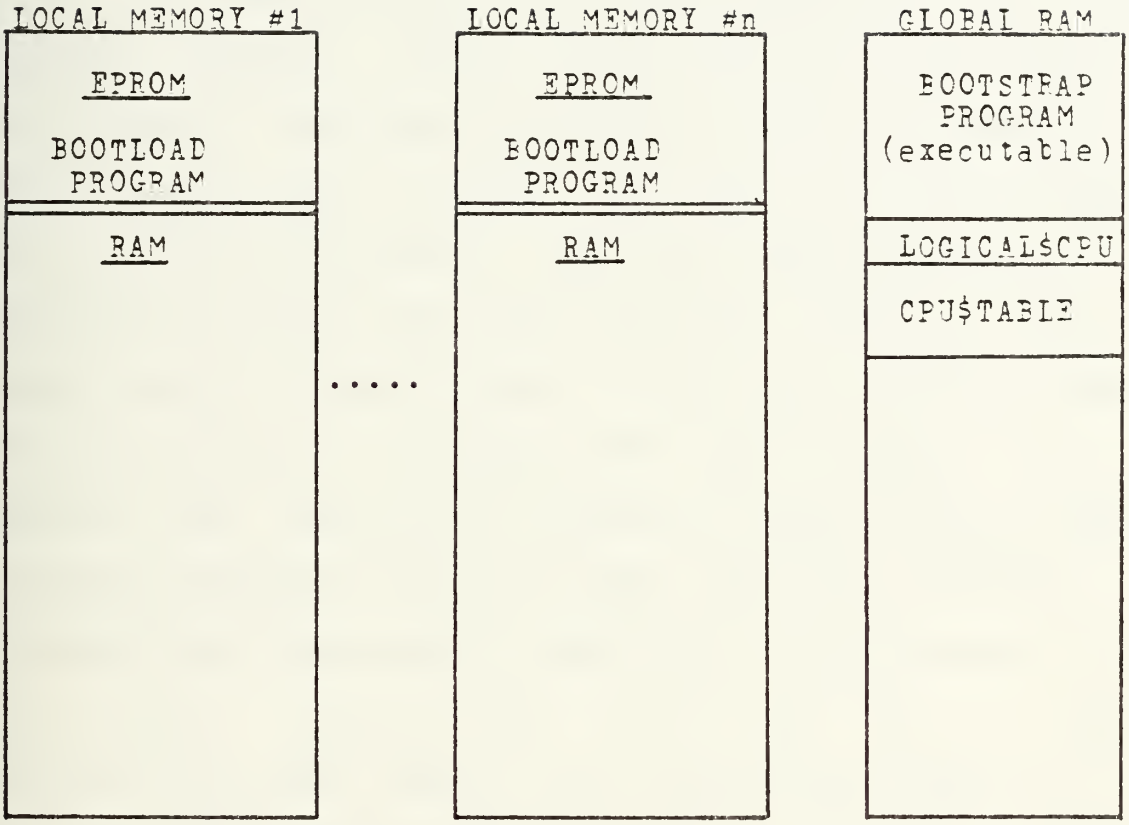
3. Loading the Bootstrap Program

With these preliminaries out of the way, the Bootload CPU can start the actual bootstrap loading function. This load involves the first access to disc by the initialization mechanism. Since the bootload program is EPROM-resident, simplicity is a primary concern. For that reason, the bootload program will merely read from a fixed address on disc, and load the data into a fixed area of global memory. For the same reason, only the Bootload CPU will access the disc. This simplifies the bootload programs by eliminating the need for a complex synchronization method to allow the processors to share the disc. The bootload program on the Bootload CPU will merely read a single disc record, and load that record into a pre-specified global memory buffer. Note that this disc record is already in executable format (viz., not a hexadecimal file). It will then transfer control, with an unconditional jump, to the location of the first byte in the buffer. This will transfer

control from the EPROM bootload program in the Bootload CPU to the bootstrap program just read in from disc. Figure III-4 shows the contents of the system memory after the bootload program has been run.

4. Executing the Bootstrap Program

The block of data just read in contains the bootstrap program developed during system generation. Recall that this program is designed to load the base layer (kernel) of the operating system from disc into primary memory. Since each processor's local memory will contain parts of this kernel, each processor will need to execute the bootstrap program to load its kernel. For simplicity, all processors will share the same bootstrap program code, that will be located in global RAM. The Bootload CPU (at this point executing the bootstrap program) will do the actual disc read for all processors. This is consistent with the method used to accomodate the the initial hardware configuration as discussed above. The Bootload CPU will load the hexadecimal file containing the base layer of the kernel into a global memory buffer, leaving it in the hexadecimal format. The Bootload CPU, since it is already running, will then be the first processor to load the kernel into its local memory. The bootstrap program includes functions to read the hexadecimal object file (the kernel) from the global RAM buffer, convert the data to its binary (executable) representation, and load it at the addresses



System Memory at end of Bootload

Figure III-4

specified in the hexadecimal file. Recall that this load address for each segment in the kernel is made up of the segment base address in the segment base address record and the load address offset contained in the data record itself.

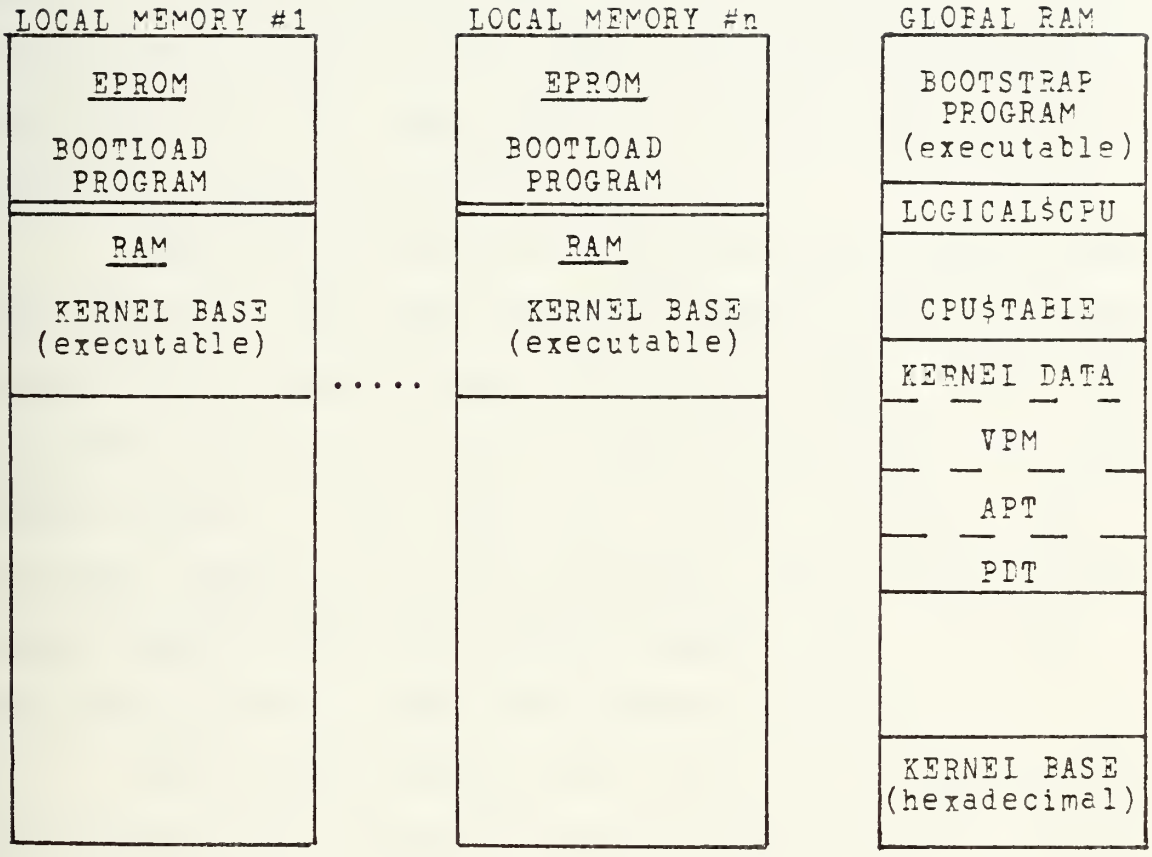
All other processors are still executing the EPROM bootload program, waiting to be signalled by the Bootload CPU via their "mailboxes". The Bootload CPU now signals each CPU in turn to load its kernel, and then waits for a signal that the CPU has done so. Note that before signalling, the Bootload CPU insures that the target CPU's kernel is in the global buffer- either read in from disc or still present from the loading of a previous CPU. When signaled by the Bootload CPU, each CPU transfers (jumps) from the EPROM bootstrap program to the global RAM bootstrap program. It then executes the routine to read the file (the kernel) from the buffer, convert the data back to its binary representation, and load it into the addresses specified in the ASCII file. Since the identity of the kernel hexadecimal file is well defined, and since the number of CPU's is known (viz., available from the CPU Table), this bootload procedure is relatively simple. Recall that simplicity is a primary goal during the bootload phase since the environment is only the bare hardware. As each processor completes its bootloading task, it will perform an unconditional jump to the first location in its kernel (now in executable form). The Bootload CPU will jump to the kernel after all other

CPU's have finished their bootloading task and signalled this fact to the Bootload CPU.

This jump will simulate a preempt interrupt in the Inner Traffic Controller interrupt handler [7]. The jump is to a special entry point in the interrupt handler routine that is used only for initialization. This entry point saves the processor register values, which include the logical and physical CPU numbers, that must be saved for later use by the Inner Traffic Controller Scheduler. The entry into the Inner Traffic Controller marks the end of the bootload phase and the transition into the run time phase. At this point, all processors are executing in the kernel. The bootstrap program is no longer needed, and will be overwritten. The system memory at the end of the bootstrap sequence is configured as shown in figure III-5.

E. RUN TIME

The loading performed at run time is conceptually quite similar to the bootloading discussed in the previous section. One difference between the two phases is that the run time loading involves all processes that are to be run. But the main difference is that the "Bootload" function is done by run time loader processes that run on the virtual processor provided by the kernel. This implies that the instruction set now includes the operating system primitives provided by the kernel (e.g. ITC_ADVANCE, ITC_AWAIT, and



System Memory at end of Bootstrap

Figure III-5

Create_Process). This provides a much more supportive environment than the bare hardware of the bootload phase.

1. Invoking the Loader Processes

To understand exactly what happens when the bootload program jumps to the preempt handler in the kernel, it will be beneficial to review just what is in the kernel base, and how the contents of the kernel go about performing the remainder of the loading activities.

There are actually two processes in the kernel base. The first is the idle virtual processor. Recall that this "processor" is invoked when there is no other useful work available to be run on a physical processor. The other kernel process is the run time loader process- just a modified version of the O'Connell and Richardson memory manager process [18]. All kernel segments are included in the address space of both these kernel processes.

The Virtual Processor Map (VPM) in the Inner Traffic Controller was initialized during system design to reflect that the idle virtual processor is "running" on each CPU. The memory manager (i.e. loader) is initialized in the ready state and with a high priority. All other virtual processors are in the idle state.

The Traffic Controller's Active Process Table (APT) is initialized with NO applications processes. All virtual

processors visible to the Traffic Controller are shown to be running an idle process.

Because of this initial state created during system generation, the jump to the Inner Traffic Controller at the end of the bootload phase appears to the kernel as a preempt interrupt of the idle virtual processor. This preempt causes the higher priority loader process to be scheduled and run on each physical processor.

These loader processes all have the Process Definition Table in their address space as an external data segment shared by all loader processes. This table is the primary data base used to drive the remainder of the loading function.

2. Loading the Application Processes

Now that the operating system kernel is running on each physical processor, it can be used to load the application processes from disc. Since each application process exists as a hexadecimal object file on the disc, and since the loader processes have a complete description of each application process in their address spaces (viz., the Process Definition Table), the remainder of the loading tasks are relatively straightforward. This will involve reading each application process from the disc, placing it, in executable (i.e. binary), form at the appropriate location in the system memory, identifying the process to

the kernel, and finally, causing the kernel to schedule and execute the application processes.

The Bootload CPU still serves as the system master, and still makes all disc I/O requests. Since their register values, including their serial numbers and logical CPU numbers, were passed to them at the beginning of run time, each processor can determine whether or not it is the Bootload CPU (i.e., is its logical CPU number Ø?). If it is not, the loader process will do an ITC_AWAIT, until it is signalled to proceed (via an ITC_ADVANCE) by the Bootload CPU. The sequence of operations performed at run time call for the Bootload CPU to read the first non-kernel hexadecimal object file from the disc and to store it in the global RAM buffer. The Bootload CPU then checks the Affinity in the Process Definition Table to determine which physical processor the process is intended to run on. It will then do an ITC_ADVANCE on the appropriate eventcount for the loader process in that CPU. Note that there is the special case of application processes being loaded on the Bootload CPU. In this case, the signalling will be slightly different. But this will require only a minor addition to the loader program.

The designated processor's loader process will load and convert the hexadecimal object file as described in the previous section. In addition, it will extract from the hexadecimal object file the CS and IP register values. It

will enter these values into this loader process's Process Parameter Block, along with the SS register value from the Process Definition Table. The loader process then calls the kernel Traffic Controller procedure "Create_Process", passing the address of the Process Parameter Block as an argument. Create_Process makes the necessary entries in the Active Process Table to describe the just-loaded process, and initializes the kernel stack for this process. Create_Process then returns into the loader process from which it was called. The loader process will, in turn, notify the Bootload CPU that it has finished, and the Bootload CPU will read in the hexadecimal object file for another process.

3. Initiating Application Process Execution

This sequence of events is repeated until the loader process on the Bootload CPU finds a null entry in the Process Definition Table, which signifies that all processes have been loaded and created. This means that all system initialization functions- system generation, bootloading, and run-time loading- are completed, and all application processes are created, loaded on their respective processors, and in the ready state. The only thing required now is for the Bootload CPU to call the ITC_SET_PREEMPT procedure for each virtual processor known to the Traffic Controller and then do an ITC_AWAIT. This will cause the

normal scheduling functions to run the highest priority process that is ready to be run on each processor.

F. SUMMARY

In this chapter, the entire sequence of events required for initialization of a multiple microcomputer system have been examined. Each of the initialization phases - system generation, bootloading, and run time - and the environments in which they occur, have been analyzed. This analysis was intended to show the reader how initialization can, indeed, be simplified by a careful sequencing of initialization activities.

IV. SUMMARY AND CONCLUSIONS

A. SUMMARY

The goal of this thesis has been to develop a system initialization mechanism for the Intel 8086-based multiple microcomputer system to be used by the Solid State Laboratory at the Naval Postgraduate School for "smart sensor" research. A secondary goal, from the outset, has been to present a system initialization design philosophy that would help fill a void in current computer science literature. This design philosophy asserts that the issues of system generation and bootstrap loading deserve a level of consideration equal to, and concurrent with, operating system issues. The basic premise of the thesis is that simplification leads to a more versatile and robust design and, subsequently, to a system initialization mechanism that is easily understood and readily adaptable to a variety of hardware and operating system configurations.

The simplification in this design approach is achieved by two means. The first is a core-image driven loader. This technique involves creating a copy of the base layer of the operating system as it should appear in primary memory immediately prior to execution. This core image is then stored on some secondary storage medium. When it is desired

to initialize the system, this core image is merely loaded into primary memory and control is passed to the first instruction.

The other, and probably more meaningful, means of simplification is to carefully sequence the required initialization activities such that each is performed in the most supportive environment available. This transfers functional complexity to a phase of initialization that enjoys the most operating system and utility program support, and removes possible complexity from the bare hardware environment of the bootload program. Since the most supportive environment in this application is available at system generation time, the goal was to accomplish as many initialization activities as possible during this phase. With the assumptions (based on the application for which the system was designed) made at system generation time, this thesis was able to fully exploit this most supportive environment. In so doing, the generation of the complete core image and all memory allocation were accomplished during system generation. As the core image of each process is created, the identity of the process (viz., its address space and execution point) were encoded into the image. Thus every process in the system could be completely characterized with information contained in its core image. This capability creates a compilation-independence that is important to a general purpose initialization mechanism.

The system initialization scheme designed for this thesis makes extensive use of the operating system kernel primitives avaiable at run time. In particular, the ITC_ADVANCE and ITC_AWAIT primitives are used for interprocess communication during the loading of the application processes, and the Create_Process function is used to identify the application processes to the kernel.

B. FOLLOW-ON WORK

This thesis has scratched the surface of an extremely interesting and challenging research area. But in developing the initialization mechanism discussed here, it brought to light many follow-on research ideas. Naturally, the first follow-on work should concentrate on completing the implementation of the design presented in this thesis. The design and implementation should then be extended to automate as many of the manual functions as possible. This should include complete automation of the linking and locating processes, possible elimination of the file conversion program, and automated memory allocation as discussed by O'Connell and Richardson [18]. This would provide programmatic creation of the Process Definition Table, initial memory map, and the other system initialization data structures. This effort will require additional documentation from the Intel Corporation on the development tools and file formats discussed in Chapter II.

Recall that this thesis made several assumptions to simplify and expedite the development process. Near-term research efforts might attempt to eliminate some of these assumptions, particularly those about the static nature of the run-time environment. This would result in a more generally applicable mechanism that would be less dependent on a priori knowledge about the system configuration. In order to achieve this generality, it will be necessary to automate most of the functions that are done manually in this thesis, particularly the memory allocation. The design of this initialization mechanism is compatible with the memory allocation scheme designed by O'Connell and Richardson, and should accept such a run-time memory allocation function without major alterations.

Of immediate concern to the smart sensor research project should be the integration of the hard disc subsystem into the hardware configuration. The availability of on-line secondary storage would permit further simplification of the initialization mechanism, and remove the need for the "controlling loader".

The most challenging research area, however, is dynamic reconfiguration and its subsequent benefit- fault tolerance. These are state-of-the-art issues that are also long term goals of the smart sensor program. They are also almost mandatory for a viable, operational smart sensor platform.

C. CONCLUSIONS

The work done in this thesis has shown the feasibility of developing a simple, versatile system initialization mechanism based on a core image approach and the careful sequencing of initialization activities. The design proposed in this thesis has not been fully tested, but sufficient functions were implemented to support the basic concepts proposed. The experience with the system thus far has shown that the concepts are not difficult to put into practice, and that they do result in a simple, easy to understand mechanism for loading and starting a process on a bare machine. The design proposals developed in this thesis should prove beneficial to future initialization development efforts, even where the hardware and operating system are different.

The thesis has also confirmed the value of an operating system with explicit segments and processes, and has shown how such an operating system structure can be exploited to significantly simplify the initialization mechanism. As this structure for microcomputer operating systems becomes more widely implemented, the methods used in this thesis can be widely applied to simplify the entire system initialization process.

APPENDIX A. UTILITY PROGRAM OUTPUT

A. OBJECTIVES

This appendix is provided to further acquaint the reader with the Intel software development utility programs used in this thesis. Each program and its pertinent parameters and options will be explained, and a sample output will be provided. While these programs are Intel products, and are designed specifically for the Intellec MDS with the ISIS-II operating system, they are representative of programs provided with other computer systems. The sample outputs at the end of this appendix are based on a very simple PL/M-86 program, written only to demonstrate the development utility programs. The source code for the sample program is shown in figure A-1.

B. THE PI/M-86 COMPILER

As mentioned in Chapter II, the PI/M-86 Compiler translates the PL/M-86 source statements into 8086 machine instructions. The MODE control in the command line determines the degree of segmentation. In the sample program compilation in figure A-2, the CODE control was used to cause the compiler to list the 8086 machine code instructions generated for each PL/M-86 instruction. Note

that the lengths of all the segments produced by the compiler are listed at the end of the output.

C. THE LINK86 PROGRAM

The linker program, as discussed in Chapter II, combines the various program modules that make up a process and resolves any external references. At the same time, it adjusts the relative addresses in the module so that they are all relative to the beginning of the output module. The sample LINK86 output listing in figure A-3 shows the list of segments produced for the sample program by the Intel linker.

D. THE LOC86 PROGRAM

The locator program is used to assign physical memory addresses to the relative addresses in the linker output module. LOC86 provides several diagnostic and output format controls [20]. Diagnostic information includes a symbol table and a complete memory map, showing the results of the locator function. This information is sent to a printable disc file unless otherwise specified. Output module controls are used to control the content of the output module, the order of the segments in the module, and the assignment of physical memory locations to the segments. The controls of primary concern here are the ADDRESSES and SEGMENTS

controls. As seen in figure A-4, these controls assign a base address to each segment in the process.

The other control of interest during system initialization is the SEGSIZE control. It is used to specify the size of one or more segments in the output module. This control is used during system generation to build the kernel stack frame discussed in Chapter III.

The sample LOC86 output in figure A-4 includes the process's symbol table and memory map. For illustrative purposes, the SEGSIZE control was used to add 20H bytes to the size of the stack segment.

E. THE OH86 PROGRAM

The final utility program used during system generation is the file conversion program, OH86. Recall that this program translates the binary object file (for which very little documentation is available) into an ASCII hexadecimal object file (which is very well documented). The sample output from OH86 is shown in figure A-5. The blank spaces and line numbers were added to improve readability, and do not occur in the actual output file.

Each hexadecimal file produced by OH86 is made up of four different record types. These record types are explained below.

1. Record Type ØØ is the Data Record. These records contain the actual program code and data that make up each process.

2. Record Type Ø1 is the End-of-File Record.

3. Record Type Ø2 is the Extended Address Record. This record specifies the segment base address for the type ØØ records that immediately follow it. For example, the type Ø2 record in line 11 of figure A-5 contains the segment base address (Ø1ØØH) for the type ØØ records in lines 12 through 18.

4. Record Type Ø3 is the Start Address Record. It specifies the Code Segment and Instruction Pointer register values for the first instruction in the Code Segment. In the example, the CS register value is Ø1ØØH, and the IP register value is ØØØ8H. The locations from the address specified in LOC86 (1ØØØH) to the address specified in the Start Address Record (1ØØ8H) are used by the compiler to store the addresses of external data segments, and the DS and SP register values (see lines Ø1 through Ø4).

Each of the records in the hexadecimal object file consists of several fields. These fields, and their effect on the loading function is explained below.

1. The Record Mark Field is used as a record delimiter. OH86 uses an ASCII colon (Ø3AH) to signify the beginning of each record.

2. The Record Length Field contains two ASCII digits that specify the length, in bytes, of the data or information contained in the record.

3. The Load Address Field contains the address offset from the segment base address (in the type 02 record) for the first data byte in the record. Note that only type 00 records have load addresses other than 0000. Recall from Chapter II that there is no boundary check made when addressing into a segment. The exact load address for a particular data byte can be calculated as follows:

EFF. ADDR. = BASE ADDRESS + [(DRLA + DRI) MODULO 64K]

Where DRLA is the Data Record Load Address, and DRI is the byte index within the Data Record.

4. The Record Type Field specifies the type of the record, as described above.

5. The Data Field contains the actual data to be converted to binary and loaded into primary memory. This is a variable length field that may be from 2 to 10H bytes long.

6. The Checksum Field is used for error detection in the loading and translating process. It contains the twos complement of the 8-bit sum of the bytes that result from converting the ASCII bytes back into binary.

SUMMARY

This appendix was intended to acquaint the user with more details concerning the software development utility programs used to develop the system generation mechanism described in this thesis. It has provided a very simple PL/M-86 program and the output from each of the development utilities. The reader desiring additional information about these programs should refer to the MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users [20].

SOURCE LISTING

```
/*****************************************************************/
/*                                                               */
/*       Sample Program to demonstrate the software              */
/*       development utility programs used during                */
/*       system generation.  This program simply                 */
/*       increments a global array element nine                  */
/*       times and then prints the result on the                 */
/*       terminal screen.                                        */
/*                                                               */
/*****************************************************************/


COUNTER1:   DO;

     DECLARE  I           BYTE,  /*loop index*/
              ARRAY(2)    BYTE EXTERNAL, /*external array*/
              PROMPT(*)   BYTE INITIAL('VALUE IS:  '),
              STATUSPORT        LITERALLY 'ØD8H',
              DATAPORT          LITERALLY 'ØDAH',
              XMITRDY           LITERALLY 'ØØ1H';

/*****************************************************************/
/*                                                               */
/*      OUTCHAR is a procedure which tests the                   */
/*      status of the serial I/O port that is                    */
/*      connected to the terminal.  If the port                  */
/*      is "ready", an ASCII character is output                 */
/*      to the CRT screen.                                       */
/*                                                               */
/*****************************************************************/

OUTCHAR:  PROCEDURE (CHAR);
     DECLARE  CHAR  BYTE;
     DO WHILE (INPUT(STATUSPORT) AND XMITRDY) = Ø;
         END; /* wait until ready to transmit*/
     OUTPUT(DATAPORT) = CHAR AND Ø7FH;
     END;   /* of OUTCHAR declaration */

     ARRAY(Ø) = Ø;  /* initialize sum */
```

PL/M-86 Source Listing

Figure A-1

```
 DO I = Ø TO 9;
    /* increment the sum */
    ARRAY(Ø) = ARRAY(Ø) + 1;
 END;  /* of DO loop */

DO I = Ø TO LAST(PROMPT);
   /* print the header */
   CALL OUTCHAR(PROMPT(I));
END;  /* of print loop */

CALL OUTCHAR(ARRAY(Ø));  /* print the sum */

END;  /* COUNTER1 program */
```

PL/M-86 Source Listing

Figure A-1 (cont'd)

```
PLM-86 COMPILER    COUNTER1


ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE COUNTER1
OBJECT MODULE PLACED IN :F1:CNTR1.OBJ
COMPILER INVOKED BY:  PLM86 :F1:CNTR1.SRC CODE LARGE
     DATE(1 JUNE 80)


/*************************************************************/
/*                                                           */
/*      Sample Program to demonstrate the software           */
/*      development utility programs used during             */
/*      system generation.  This program simply              */
/*      increments a global array element nine               */
/*      times and then prints the result on the              */
/*      terminal screen.                                     */
/*                                                           */
/*************************************************************/


   1          COUNTER1:   DO;

   2   1      DECLARE  I           BYTE,   /* loop index */
                  ARRAY(2)   BYTE EXTERNAL,
                  PROMPT(*)  BYTE INITIAL('VALUE IS:  '),
                  STATUSPORT       LITERALLY '0D8H',
                  DATAPORT         LITERALLY '0DAH',
                  XMITRDY          LITERALLY '001H';

/*************************************************************/
/*                                                           */
/*      OUTCHAR is a procedure which tests the               */
/*      status of the serial I/O port that is                */
/*      connected to the terminal.  If the port              */
/*      is "ready", an ASCII character is output             */
/*      to the CRT screen.                                   */
/*                                                           */
/*************************************************************/

   3   1      OUTCHAR:  PROCEDURE (CHAR);
                                        ; STATEMENT # 3
```

PL/M-86 Compiler Listing

Figure A-2

```
                        OUTCHAR      PROC NEAR
        0073  55                     PUSH    BP
        0074  8BEC                   MOV     BP,SP
4   2       DECLARE  CHAR  BYTE;
5   2       DO WHILE (INPUT(STATUSPORT) AND XMITRDY) = 0;
                END;
                                               ; STATEMENT # 5
                        @1:
        0076  E4D8                   IN      0D8H
        0078  F6C001                 TEST    AL,1H
        007B  7403                   JZ      $+5H
        007D  E90300                 JMP     @2
                                               ; STATEMENT # 6
        0080  E9F3FF                 JMP     @1
                        @2:
7   2             OUTPUT(DATAPORT) = CHAR AND 07FH;
                                               ; STATEMENT # 7
        0083  8A4604                 MOV     AL,[BP].CHAR
        0086  80E07F                 AND     AL,7FH
        0089  E6DA                   OUT     0DAH
8   2             END;
                                               ; STATEMENT # 8
        008B  5D                     POP     BP
        008C  C20200                 RET     2H
                        OUTCHAR      ENDP

9   1             ARRAY(0) = 0;  /* initialize sum */
                                               ; STATEMENT # 9
        0008  FA                     CLI
        0009  2E8E160400             MOV     SS,CS:@@STACK$FRAME
        000E  BC0600                 MOV     SP,@@STACK$OFFSET
        0011  8BEC                   MOV     BP,SP
        0013  2E8E1E0600             MOV     DS,CS:@@DATA$FRAME
        0018  FB                     STI

        0019  2EC41E0000             LES     BX,CS:@ARRAY
        001E  26C60700               MOV     ES:ARRAY[BX],0H
10  1             DO I = 0 TO 9;
                                               ; STATEMENT # 10
        0022  C606000000             MOV     I,0H
                        @3:
        0027  803E000009             CMP     I,9H
        002C  7603                   JBE     $+5H
        002E  E91100                 JMP     @4
```

PL/M-86 Compiler Listing

Figure A-2 (cont'd)

```
11   2              ARRAY(0) = ARRAY(0) + 1;
                      /* increment sum */
                                              ; STATEMENT # 11
         0031  2EC41E0000       LES      BX,CS:@ARRAY
         0036  26FE07           INC      ES:ARRAY[BX]
12   2            END;  /* DO LOOP */
                                              ; STATEMENT # 12
         0039  FE060000         INC      I
         003D  7403             JZ       $+5H
         003F  E9E5FF           JMP      @3
                        @4:

13   1           DO I = 0 TO LAST(PROMPT);
                  /* print the "header" */
                                              ; STATEMENT # 13
         0042  C606000000       MOV      I,0H
                        @5:
         0047  803E00000A       CMP      I,0AH
         004C  7603             JBE      $+5H
         004E  E91500           JMP      @6
14   2              CALL OUTCHAR(PROMPT(I));
                                              ; STATEMENT # 14
         0051  8A1E0000         MOV      BL,I
         0055  B700             MOV      BH,0H
         0057  FF7701           PUSH     PROMPT[BX]; 1
         005A  E81600           CALL     OUTCHAR
15   2           END;
                                              ; STATEMENT # 15
         005D  FE060000         INC      I
         0061  7403             JZ       $+5H
         0063  E9E1FF           JMP      @5
                        @6:
16   1           CALL OUTCHAR(ARRAY(0));
                    /* print the sum */
                                              ; STATEMENT # 16
         0066  2EC41E0000       LES      BX,CS:@ARRAY
         006B  26FF37           PUSH     ES:ARRAY[BX]
         006E  E80200           CALL     OUTCHAR

17   1           END;  /* COUNTER1 */
                                              ; STATEMENT # 17
         0071  FB               STI
         0072  F4               HLT
```

PL/M-86 Compiler Listing

Figure A-2 (cont'd)

```
MODULE INFORMATION:
     CODE AREA SIZE     = 008FH    143D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 000CH     12D
     MAXIMUM STACK SIZE = 0006H      6D
     52 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

PL/M-86 Compiler Listing

Figure A-2 (cont'd)

LINK86 LISTING

```
ISIS-II MCS-86 LINKER, V1.1, INVOKED BY:
LINK86 :F1:CNTR1.OBJ, :F1:ARRAY.OBJ TO :F1:CNTR1.LNK
LINK MAP FOR :F1:CNTR1.LNK(COUNTER1)

LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS  SEGMENT          CLASS
 008FH  ------  COUNTER1_CODE    CODE
 00CEH  ------  COUNTER1_DATA    DATA
 0006H  ------  STACK            STACK
 0000H  ------  MEMORY           MEMORY
 0000H  ------  ARRAYDEC_CODE    CODE
 0002H  ------  ARRAYDEC_DATA    DATA

INPUT MODULES INCLUDED:
:F1:CNTR1.OBJ(COUNTER1)
:F1:ARRAY.OBJ(ARRAYDEC)
```

LINK86 Listing

Figure A-3

```
ISIS-II MCS-86 LOCATER, V1.1 INVOKED BY:
LOC86 :F1:CNTR1.LNK TO :F1:CNTR1.RUN ADDRESSES(SEGMENTS&
(COUNTER1_CODE(1000H),COUNTER1_DATA(2000H),STACK(3000H),&
ARRAYDEC_DATA(30000H),ARRAYDEC_CODE(31000H),&
MEMORY(31100H)))&
SEGSIZE(STACK(+20H)) RS(0 TO 0FFFH)

SYMBOL TABLE OF MODULE COUNTER1
READ FROM FILE :F1:CNTR1.LNK
WRITTEN TO FILE :F1:CNTR1.RUN

BASE   OFFSET TYPE SYMBOL          BASE   OFFSET TYPE SYMBOL

3000H  0000H  PUB  ARRAY

ARRAYDEC:  SYMBOLS AND LINES
3110H  0000H  SYM  MEMORY          3000H  0000H  SYM  ARRAY
3100H  0000H  LIN       3

MEMORY MAP OF MODULE COUNTER1
READ FROM FILE :F1:CNTR1.LNK
WRITTEN TO FILE :F1:CNTR1.RUN

MODULE START ADDRESS  PARAGRAPH = 0100H  OFFSET = 0008H
SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

01000H   0108EH      008FH   W   COUNTER1_CODE     CODE
02000H   020CBH      00CCH   W   COUNTER1_DATA     DATA
03000H   03025H      0026H   W   STACK             STACK
30000H   30001H      0002H   W   ARRAYDEC_DATA     DATA
31000H   31000H      0000H   W   ARRAYDEC_CODE     CODE
31100H   31100H      0000H   W   MEMORY            MEMORY
```

LOC86 Listing

Figure A-4

```
01  : 02 0000 02 0100 FB
02  : 04 0000 00 0000 0030 CC
03  : 02 0000 02 0100 FB
04  : 04 0004 00 0003 0002 F3
05  : 02 0000 02 0200 FA
06  : 10 0002 00 474C4F42414C2056414C55452049533A AA
07  : 02 0012 00 2020 AC
08  : 02 0000 02 0107 F4
09  : 10 0003 00 558BECE4D8F6C0017403E90320E9F3FF 70
10  : 0C 0013 00 8A460480E07FE6DA5DC20200 4D
11  : 02 0000 02 0100 FB
12  : 10 0008 00 FA2E8E160400BC06008BEC2E8E1E0600 FF

17  : 10 0058 00 7702E81600FE0600007403E9E1FF2EC4 EB
18  : 0B 0068 00 1E000026FF37E80200FBF43A
19  : 04 0000 03 01000008F0
20  : 00 0000 01 FF
```

OH86 Listing

FIGURE A-5

LIST OF REFERENCES

1. Naval Postgraduate School, Abstract 178-81, Focal Plane Processing Techniques for Background Clutter Suppression and Target Detection, Tao, T. F., Hilmers, D., Evenor, B., Bar-Yehoshua, D., August 1979

2. Bar-Yehoshua, D., Two Dimensional Non-recursive Filter for Estimation and Detection of Targets, E. E. Thesis, Naval Postgraduate School, June 1977

3. Evenor, B., Statistical Non-recursive Spatial-Temporal Focal Plane Processing for Background Clutter Suppression and Target Detection, Ph. D. Thesis, Naval Postgraduate School, December 1977

4. Hilmers, D., Spatial-Temporal Filter for Clutter Suppression and Target Detection of Real World Infrared Images, M. S. Thesis, Naval Postgraduate School, December 1978

5. Celik, K., Focal Plane Signal Processing for Clutter Suppression and Target Detection in Infrared Images, E. E. Thesis, Naval Postgraduate School, June 1979

6. Brenner, R., Multiple Microprocessor Architecture for Smart Sensor Focal Plane Image Processing, M. S. Thesis, Naval Postgraduate School, December 1979

7. Wasson, W. J., Detailed Design of the Kernel of a Real Time Multiprocessor Operating System, M. S. Thesis, Naval Postgraduate School, June 1980

8. Madnick, S., and Donovan, J., Operating Systems, McGraw Hill, 1974

9. Schell, R. R., Dynamic Reconfiguration in a Modular Computer System, Ph. D. Thesis. M.I.T., May 1971

10. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Sc. D. Thesis, M.I.T., June 1966

11. Luniewski, A., A Simple and Flexible System Initialization Mechanism, M. S. Thesis, M.I.T., May 1977

12. System Development Corporation Report TM-3225, Management Handbook for Estimation of Computer Programming Costs, by E. A. Nelson, 1974

13. Corbató, F., Saltzer, J. and Clingen, C., Multics- The First Seven Years, AFIPS Proceedings of the Spring Joint Computer Conference, vol. 40, 1972

14. Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975

15. The 8086 Family User's Manual, 9800722-03, Intel Corp., October 1979

16. The ICE-86 In Circuit Emulator User's Guide, 9800XXXX, October 1979

17. Organick, S., Multics: An Examination of Its Structure, M.I.T. Press, 1972

18. O'Connell, J., and Richardson, D., Secure Design for a Multi-Microprocessor Operating System, M. S. Thesis, Naval Postgraduate School, June 1979

19. Reed, D., and Kanodia, R., "Synchronization with Eventcounts and Sequencers", Communications of the ACM, v. 22, p. 115-123, February 1979

20. MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, 9800639-03, Intel Corp., 1979

21. ISIS-II PL/M-86 Compiler Operator's Manual, 9800478-03 Rev C, Intel Corp., 1980

22. PL/M-86 Programming Manual, 9800466A, Intel Corp., 1978

23. Peripheral Design Handbook, 9800676B, Intel Corp., 1979

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Mr. Joel Trimble Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Asst. Professor U. R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Lt. Col. R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
7.	Professor T. F. Tao, Code 62Tv Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	2
8.	Lt Warren J. Wasson, USN Commander Naval Electronics Systems Command PME 124 Washington, D.C. 20360	1
9.	Capt John L. Ross 552 AWACW/ADM Tinker AFB, Oklahoma 73145	3

Thesis
R77565 Ross
c.1

189902

Design of a system initialization mechanism for a multiple microcomputer.